Privacy-Preserving Constrained Evaluation of LLM-Generated HLS C/C++

Nuo Xu
University of Minnesota - Twin Cities
Minneapolis, MN, USA
xu001536@umn.edu

Jinwei Tang
University of Minnesota - Twin Cities
Minneapolis, MN, USA
tang0940@umn.edu

Zihang Chen
Simon Fraser University
Burnaby, BC, Canada
zihang_chen_3@sfu.ca

Xiaolin Xu
Northeastern University
Boston, MA, USA
x.xu@northeastern.edu

Wujie Wen
North Carolina State University
Raleigh, NC, USA
wwen2@ncsu.edu

Zhenman Fang
University of Minnesota - Twin Cities
Minneapolis, MN, USA
zhenman@umn.edu

Caiwen Ding
University of Minnesota - Twin Cities
Minneapolis, MN, USA
dingc@umn.edu

Abstract

Large language models (LLMs) are increasingly evaluated on hardware design tasks through syntax checks, simulation, HLS synthesis, and performance, power, and area (PPA) metrics. These metrics are necessary for HLS usability, but they do not answer whether generated accelerators satisfy security-relevant source-level constraints. This paper studies that missing layer for HLS C/C++ generation and repair. We construct a preliminary security-constrained evaluation overlay for 17 primary HLS kernels by adding source-level policies, deterministic policy checking, functional harnesses where available, and Vitis HLS synthesis for pre-HLS-passing candidates; six NTT/AutoNTT tasks are additionally reported as G1/G3 source-level extension cases. The evaluation compares six descriptive generation and repair conditions: Spec-only generation, Policy-conditioned generation, Rule-aware generation, Checker-feedback repair, Strategy-guided generation, and Semantic-feedback repair. A separate GateRepair case study repairs simple cases but does not dominate one-call settings on hard ciphers. Across eight stable model conditions, pre-HLS pass ranges from 23.5% to 37.9%, and the two best one-call settings are Checker-feedback repair and Semantic-feedback repair. The results expose two distinct mismatch patterns: candidates can pass functional tests while violating the source-level policy, or pass the policy checker while failing functionality. Completed HLS runs synthesize most pre-HLS-passing candidates, but PPA examples show large latency or area shifts. These results show that security-constrained LLM-HLS evaluation must separate syntax, functionality, source-level policy compliance, HLS synthesis, and PPA evidence.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; • **Security and privacy →** *Hardware security implementation.*

GLSVLSI '26, Canandaigua, NY, USA

ACM ISBN 979-8-4007-2431-2/26/06
https://doi.org/10.1145/3787109.3816406

Keywords

high-level synthesis, large language models, hardware security, source-level policy, validation

ACM Reference Format:
Nuo Xu, Jinwei Tang, Zihang Chen, Xiaolin Xu, Wujie Wen, Zhenman Fang, and Caiwen Ding. 2026. Privacy-Preserving Constrained Evaluation of LLM-Generated HLS C/C++. In *Great Lakes Symposium on VLSI 2026 (GLSVLSI '26), June 22–24, 2026, Canandaigua, NY, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3787109.3816406

1 Introduction

LLM-assisted HLS generation, repair, and design-space exploration are becoming a crowded evaluation area. Recent benchmarks and systems evaluate whether an LLM can produce C/C++ that parses, passes software simulation, synthesizes under HLS tools, and improves PPA through report-driven repair or directive tuning [1, 6, 8, 10, 16]. These gates are necessary for generic HLS productivity, but they are not sufficient for security-sensitive accelerators. A cryptographic or security-sensitive kernel must also avoid source-level structures that encode timing, leakage, or Trojan-like behavior.

The missing layer appears before RTL exists. HLS C/C++ can contain secret-dependent branches, data-dependent loop exits, table or pointer addresses derived from secret data, secret-tainted public outputs, or rare trigger patterns. These constructs can be security-relevant even when the code compiles, passes a functional test, and enters synthesis. Conversely, a rewrite that removes a flagged source-level pattern may break the algorithm or create a hardware design with unacceptable PPA. Security-related LLM-HLS generation therefore needs an evaluation protocol that separates ordinary HLS validity from source-level policy compliance.

This paper evaluates current LLM capability on security-related HLS C/C++ generation and repair. Starting from inspected HLS kernels, we add explicit source-level policies, checker expectations, functional harnesses where available, and HLS synthesis wrappers. We then ask models to generate or repair complete HLS candidates under different information and feedback conditions. Every candidate is judged by deterministic syntax, functional, source-level policy, and HLS synthesis gates. We also report six NTT/AutoNTT extension tasks as source-level G1/G3 stress cases and treat GateRepair as a separate multi-call case study rather than a same-budget one-call setting.

The central failure mode is not only that generated code may be syntactically invalid or functionally wrong. A candidate can also be functionally plausible and synthesizable while still violating a source-level policy. We therefore use *source-level policy pass* (SL-policy pass) as a distinct metric. The metric records whether the checker finds violations of the stated source-level policy under the task annotations; it is not a claim of RTL noninterference, physical side-channel resistance, or complete hardware security.

This paper contributes a preliminary capability evaluation for security-related HLS C/C++ generation and repair, a security-policy overlay with deterministic validation for 17 primary HLS kernels, and multi-gate empirical evidence showing that syntax, functionality, SL-policy compliance, HLS synthesis, and PPA are distinct evidence levels that should not be collapsed into one pass rate.

2 Background and Related Work

2.1 Executable LLM-HLS Evaluation

LLM-HLS benchmarks provide staged evaluation procedures for code generation, simulation, synthesis, and PPA reporting [1, 6]. They establish that HLS generation must be judged by executable artifacts rather than natural-language plausibility alone. HLS-Eval is closest to a reusable Vitis-style harness for LLM-generated HLS C/C++; Bench4HLS emphasizes end-to-end compilation, simulation, synthesis, and PPA reporting. These works define the standard HLS usability gates that our evaluation reuses: syntax, functional execution, synthesis, and hardware-quality evidence.

Repair and optimization systems build on the same tool-backed principle. HLS-Repair targets C/C++ program repair for HLS; ChatHLS and TimelyHLS use feedback from HLS tools, timing reports, or architecture-specific constraints to improve candidate designs [8, 10, 16]. LLM-DSE and iDSE use LLMs as structured search policies over HLS directive spaces rather than as final correctness authorities [9, 14]. Agentic-HLS similarly shows that LLM reasoning is most useful when grounded in HLS-specific evidence rather than left as an unconstrained predictor [11]. These systems support our use of deterministic validators, but they primarily optimize HLS validity, QoR, or timing rather than source-level security-policy compliance.

2.2 Hardware-Security and LLM-Security Benchmarks

Hardware-security and LLM-security benchmarks provide the complementary security context. HardSecBench, SecV, SecFSM, HarmChip, Trust-Hub-style Trojan resources, and related work organize hardware weaknesses, RTL/Verilog tasks, finite-state-machine settings, Trojan patterns, and security-aware prompting setups [2–4, 15]. They show that hardware security is an important LLM-evaluation target and provide useful taxonomies for trigger, leakage, and access-control behavior. However, they do not directly provide the combined artifact needed here: Vitis-HLS C/C++ generation tasks with functional harnesses, synthesis/PPA flow, and explicit source-level security policies.

2.3 HLS Security Methods

Prior HLS-security research motivates the policy classes used in this evaluation. Information-flow, taint, timing-sensitive control, constant-time design, and security-aware HLS methods show why source-level control flow, memory addresses, sinks, and trigger patterns are security-relevant [5, 7, 12, 13]. ASSURE-style timing-sensitive information-flow enforcement motivates explicit secret/public labels and control-dependence checks. Constant-time HLS work motivates source-level treatment of data-dependent latency. SecHLS and TaintHLS motivate security-aware scheduling/binding constraints and taint propagation. This paper does not replace those downstream analyses. Instead, it evaluates whether LLM-generated HLS C/C++ candidates can pass an explicit source-level policy layer before synthesis and PPA evidence are interpreted.

2.4 Combined Gap

Taken together, these lines of work define the gap this paper targets. Existing HLS benchmarks provide executable tasks and synthesis metrics. Existing hardware-security benchmarks provide threat classes and security prompts. HLS-security methods provide semantic motivation. The missing middle is a security-constrained LLM-HLS evaluation flow that combines executable HLS tasks, source-level security labels, deterministic checking, functional testing, and synthesis/PPA validation.

3 Evaluation Protocol: Security-Constrained HLS Tasks and Gates

3.1 Evaluation Unit and Policy Target

The basic unit of evaluation is one complete HLS C/C++ candidate produced for a fixed task by a fixed model condition under one one-call setting. The model may generate a candidate from a specification or repair a candidate from feedback, but the candidate is evaluated only by deterministic validators. This unit is used consistently in the denominator definitions in Section 4.

The security-specific target is source-level policy compliance, abbreviated as SL-policy compliance in the tables. Each task policy marks secret inputs, public inputs and outputs, allowed declassification, and forbidden source-level flows. The checker reports violations of secret-controlled branches, secret-controlled loops or exits, secret-derived memory addresses, tainted public/debug/status sinks, and task-specific known violation patterns. An SL-policy pass means that the checker reports zero findings for the generated source under the task policy.

These are task-specific policies, not a generic natural-language safety prompt. For AES/PRESENT-style tasks, state and key bytes are secret, the final ciphertext may be declassified, and secret-indexed S-box/table addresses are forbidden. For statistical-test and hash tasks, the input stream is treated as secret or security-sensitive, the final statistic or digest may be declassified, and tainted debug/status outputs or trigger-like predicates are forbidden. For NTT/ AutoNTT tasks, coefficient memories or streams are secret, twiddle factors and transform controls are public, the final NTT/ INTT result may be declassified, and coefficient-dependent control, shuffle addresses, FIFO decisions, and status leaks are forbidden.

This target is intentionally scoped. SL-policy pass is not an RTL noninterference proof, a physical side-channel claim, or a complete hardware-security guarantee. For cryptographic tasks, the policy can allow the specified functional output value to be declassified while still treating source-level secret-dependent table addresses as violations. Thus an AES, DES, or PRESENT candidate can be functionally correct while failing the strict source-level policy.

Table 1: One-call generation and repair conditions. All settings use one model call and the same primary denominator; the table describes experimental prompt conditions, not new frameworks.

Setting	Prompt condition	Question tested
Spec-only generation	Task specification and fixed HLS function signature only	Baseline G1–G3 behavior without policy information
Policy-conditioned generation	Spec-only prompt plus the task-specific source-level policy	Whether explicit policy text improves SL-policy compliance without destroying functionality
Rule-aware generation	Policy-conditioned prompt plus general HLS/security coding rules	Whether general rules help translate the policy into source-level implementation choices
Strategy-guided generation	Policy-conditioned prompt plus a constructive secure-implementation strategy	Whether a concrete coding strategy helps beyond policy text and general rules
Checker-feedback repair	Candidate source plus one deterministic checker report	Whether the first reported violation is actionable in a single repair call
Semantic-feedback repair	Checker report plus semantic repair guidance and strategy	Whether one-call repair addresses the tainted-flow cause rather than only the surface pattern

3.2 Validation Gates and Metrics

We use *gate* to mean one deterministic validation step in the candidate pipeline. The ordered gates are G1–G4: G1 checks whether the generated HLS C/C++ source can be parsed and built by the project harness; G2 checks functional correctness when a task harness is available; G3 checks SL-policy compliance with the source-level policy checker; and G4 runs Vitis HLS synthesis on candidates that have already passed the pre-HLS checks. The gate numbers identify validation order, not increasing security levels. G1, G2, and G4 measure source validity, algorithmic validity, and hardware realizability; G3 is the source-level security-policy evidence. In all result tables, SL-policy abbreviates this G3 check: the candidate has zero findings from the task policy checker. It does not mean that the design has been proven secure.

We report two combined metrics. A pre-HLS pass means that the candidate passes G1, passes G2 when a functional harness is available, and passes G3. A final all-gate pass additionally requires G4. PPA is interpreted only for candidates whose functional and SL-policy status makes the synthesized hardware result meaningful.

3.3 One-Call Generation and Repair Conditions

Table 1 defines the six one-call settings. They are not new frameworks, but a controlled set of generation and repair conditions organized as four generation settings and two one-call repair settings. The generation settings vary the information provided before the model writes code; the repair settings provide a fixed validation report in one follow-up call. All six settings have the same call budget and the same primary denominator, so differences in the results can be attributed to the prompt condition rather than to more attempts. The settings are diagnostic, not monotonic: stronger security guidance can improve G3 while hurting G2.

Checker-feedback and Semantic-feedback are repair-style inputs, but they are still one-call settings: the model receives a fixed repair prompt once, and the repaired candidate is evaluated once. They therefore share the same denominator as the generation settings in the main setting-level aggregate.

3.4 Multi-Call GateRepair Case Study

GateRepair is separate from the one-call settings. It closes the validation loop: after each attempt, the candidate is evaluated, the first failing gate is identified, that failure is fed back, and the model returns a complete revised source file. Because this uses multiple calls and a changing prompt trajectory, it has a different cost model and is reported only as a case study. GateRepair therefore answers a different question from Semantic-feedback: whether iterative deterministic feedback can move a candidate through the gate stack after a one-call attempt fails.

4 Evaluation

4.1 Setup

Workloads. We instantiate the protocol on 17 primary HLS C/C++ tasks selected to stress different source-level policy patterns. The primary workload contains AES components and AES-style kernels, other block ciphers, hash kernels, and statistical-test kernels. Concretely, the primary pool consists of nine AES/component tasks (SubBytes, AddRoundKey, ShiftRows, MixColumns, block AES, AES, MachSuite AES, AES-table, and AES-tableless), two other block ciphers (DES and PRESENT), two hash kernels (SHA-256-WIP and CHStone-SHA), and four statistical-test kernels (cusums, monobit, overlapping, and runs). Each primary task package includes a fixed interface, task specification, source-level policy annotations, checker expectations, and the available functional and HLS harnesses used by the deterministic gates. The mix is intentional: block-cipher tasks expose secret-dependent tables and control/data-flow patterns, hash tasks stress longer dataflow kernels, and randomness tests provide smaller security-relevant kernels where functional correctness is easier to isolate.

We also include six NTT/AutoNTT extension tasks as source-level stress cases: four AutoNTT full-kernel variants with different modular-reduction styles (naive, Montgomery, Barrett, and WLM) and two HLSFactory/AutoNTT integration cases (a butterfly unit and a full NTT kernel). Their policies mark coefficient memories/streams as secret, treat twiddle factors and transform controls as public, allow only the final NTT/INTT result as declassified output, and forbid coefficient-dependent control, memory/shuffle addresses, stream decisions, and debug/status leaks. We keep these rows outside the primary all-gate denominator until matching functional and HLS wrappers are available.

Run matrix and denominators. For each model condition, the primary matrix contains 17 tasks evaluated under the six one-call settings from Section 3, giving 102 primary rows. Functional pass rates use the 96-row subset with automatic functional harness coverage. Six NTT/AutoNTT tasks are generated under the same six settings, producing 36 additional case-study rows per model condition. They use the same LLM settings and the same G1/G3 source-level checking flow as the 17 primary tasks; the difference is coverage of the downstream gates. The primary tasks have the project-local functional/HLS harness coverage needed for the pre-HLS and HLS tables, while the NTT/AutoNTT rows currently do not. The expanded generation log therefore has 138 rows per model condition, but primary all-gate results use the 102-row primary scope and NTT/AutoNTT is reported separately as G1/G3 extension evidence.

Table 2: Stable primary model-condition results. Gate columns are percentages. Syntax, SL-policy, and pre-HLS use 102 primary rows; functional and mismatch columns use the 96-row subset with functional harnesses. Setting columns report pre-HLS pass counts out of 17 primary tasks and replace the model-by-setting heatmap. Main fail is the largest mutually exclusive first-failing bucket out of 102 rows. DeepSeek gate values are three-trial means; Gemma4 and Qwen3.5 rows are current single-trial values.

		Gate rates (%)						Pre-HLS by setting (count/17)							
Model	Reasoning	Syn	Func	SLP	Pre	F+/SL-	SL+/F-	Spec	Policy	Rules	Check	Strat	Sem	Main fail	HLS
Gemma4	disabled	84.3	42.7	65.7	33.3	11.5	34.4	3	4	6	7	6	8	Func 39	33/34
Gemma4	enabled	86.3	41.7	67.6	32.4	9.4	35.4	2	3	5	9	4	10	Func 44	33/33
DS-V4-Flash	disabled	82.7	43.0	63.0	28.4	16.7	37.5	2	3	4	5	5	9	Func 36	27/28
DS-V4-Flash	enabled	81.4	40.6	74.8	34.6	6.3	41.7	2	6	5	9	4	8	Func 39	33/34
DS-V4-Pro	disabled	85.3	34.4	62.5	23.5	9.4	37.5	1	4	3	6	4	5	Func 52	23/23
DS-V4-Pro	enabled	82.4	40.9	78.4	37.9	4.2	39.6	2	4	6	13	6	13	Func 35	42/44
Qwen3.5-122B-A10	disabled	62.7	31.3	63.7	23.5	6.3	38.5	1	3	3	7	2	8	Syn 38	32/33
Qwen3.5-122B-A10	enabled	83.3	36.5	68.6	26.5	10.4	44.8	3	3	4	8	4	5	Func 46	40/46

Model conditions. The stable main comparison uses eight model conditions: Gemma4, DeepSeek Flash, DeepSeek Pro, and Qwen3.5-122B-A10, each with reasoning disabled and enabled. Each model condition is evaluated over the same primary task matrix and one-call setting set. DeepSeek pass@3 trials are treated as supplementary robustness evidence, not as the main single-run comparison. GateRepair is evaluated separately because it uses a multi-call feedback loop rather than the one-call budget used by the primary setting aggregate.

Records and result mapping. Each run records the task, task group, model condition, setting, raw model response, extracted source, syntax result, functional result when available, SL-policy checker report, first failing gate, and HLS synthesis result when attempted. The result subsections follow this accounting. The model-condition table compares the overall gate outcomes; the setting-level aggregate compares the 4+2 one-call condition set; the mismatch columns isolate candidates that pass functionality but fail the source-level policy, or pass the policy checker while failing functionality; task-group analysis shows where failures concentrate; HLS/PPA reporting is restricted to pre-HLS-passing candidates; and GateRepair is reported only as a multi-call case study.

Claim map. The evaluation is organized around six claims. First, security-sensitive HLS needs an SL-policy gate in addition to syntax, functional, and synthesis gates. Second, current one-call LLM generation remains weak under the combined pre-HLS gate. Third, functional correctness and SL-policy compliance are orthogonal failure modes. Fourth, the information ladder is not monotonic: more prescriptive security guidance can improve policy compliance while hurting functionality. Fifth, task family matters, with cipher semantics dominating the hard cases even under iterative repair. Sixth, synthesis success does not settle hardware quality because PPA can shift by orders of magnitude.

4.2 Overall Gate Results

Table 2 reports the stable primary model-condition rows. The main outcome is low multi-gate yield: pre-HLS pass ranges from 23.5% to 37.9% across the reported rows. DeepSeek Pro reasoning-enabled is the strongest reported row, but it still passes fewer than half of the 102 primary rows. The 14.4-point spread between the weakest and strongest rows shows that model choice matters, but no model condition is close to solving the task set. The mismatch columns

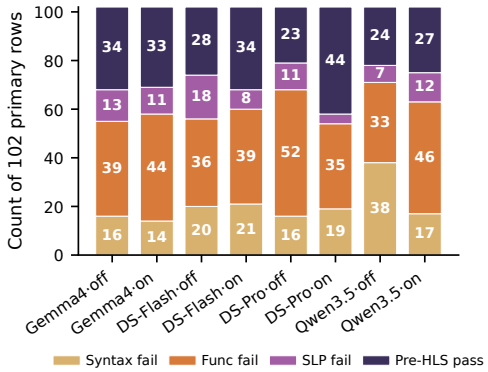

Figure 1: First-failing-gate decomposition over 102 primary rows per model condition. The mutually exclusive buckets show whether each row first fails syntax, functionality, source-level policy, or passes all pre-HLS gates.

show why this result cannot be explained by one weak validator: some candidates are functional but violate the source-level policy, while many others satisfy the policy checker but fail the functional harness. This is the core reason to separate the gates: syntax and HLS synthesis alone would overstate usable security-constrained generation, while the SL-policy checker alone would overstate candidates that no longer implement the intended algorithm.

The setting-count columns show that Checker-feedback and Semantic-feedback are the only settings that regularly push a model-setting cell near or above half of the 17 tasks; Spec-only generation remains weak across models. Figure 1 expands the compact Main fail column into a first-failing-gate decomposition: functional failure is the largest blocker for seven of eight model conditions, while Qwen3.5 reasoning-disabled is dominated by syntax failure. Reasoning mode is not a uniform win: it improves Qwen3.5 syntax substantially, gives DeepSeek Pro the best aggregate row, but does not improve Gemma4's overall pre-HLS rate. The per-setting columns make this visible without a separate model-by-setting figure.

DeepSeek repeated trials give a robustness check rather than a replacement for the canonical rows. For DeepSeek Pro reasoning-enabled, trial-mean pre-HLS is 38.7%, while any-pass-over-three coverage reaches 55/102. DeepSeek Flash reasoning-enabled shows the same gap, 34.6% trial-mean versus 50/102 coverage. Thus pass@3 is useful evidence about sampling variance, but it is not the same quantity as average one-call success.

Table 3: Setting-level aggregate across the eight model-condition rows. Columns are percentages. Syntax, SL-policy, and pre-HLS use 136 rows per setting; functional and mismatch use 128 rows with functional harness coverage.

Setting	Syntax	Functional	SL-policy	Pre-HLS	Func+/SL-	SL+/Func-
Spec-only	76.5%	29.7%	51.5%	11.8%	21.1%	39.8%
Policy-conditioned	77.2%	28.1%	69.1%	22.1%	7.8%	47.7%
Rule-aware	81.6%	27.3%	69.9%	26.5%	3.9%	44.5%
Checker-feedback	87.5%	63.3%	67.6%	47.1%	14.8%	21.9%
Strategy-guided	76.5%	27.3%	73.5%	25.7%	2.3%	49.2%
Semantic-feedback	82.4%	55.5%	75.7%	48.5%	5.5%	28.9%

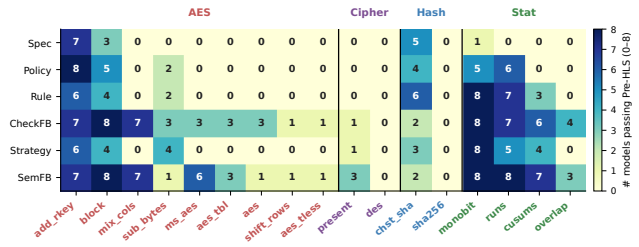

Figure 2: Setting-by-task pre-HLS coverage. Each cell reports how many of the eight stable model conditions pass all pre-HLS gates for that setting-task pair; darker cells indicate broader success.

4.3 One-Call Setting Results

Table 3 summarizes the six one-call settings. Three points matter. First, adding policy and general rules improves SL-policy pass but barely moves functional pass. Second, Strategy-guided generation maximizes SL-policy pass but produces the largest SL+/Func- mismatch, so more prescriptive security guidance can break the algorithm. Third, Checker-feedback and Semantic-feedback give the best pre-HLS rates because concrete validator output is more actionable than policy text alone. The numbers expose the non-monotonicity directly: Policy-conditioned prompts raise SL-policy pass from 51.5% to 69.1%, but functional pass stays below 30%; Strategy-guided prompts reach 73.5% SL-policy pass yet peak at 49.2% SL+/Func- mismatch; Checker-feedback and Semantic-feedback are the only settings above 47% pre-HLS because they improve functionality and policy compliance together. In short, telling a model the security policy or general constant-time rules is not enough to produce usable HLS code; the model also needs feedback that anchors the policy to the generated implementation.

Figure 2 shows where the setting gains come from. Each cell counts how many of the eight model conditions pass pre-HLS for one task under one setting. The dark cells around `add_round_key`, `block`, `monobit`, and `runs` show that some tasks are reliably solvable, while `des`, `present`, and `sha256_wip` remain sparse even with feedback. This is the main task-level result: the prompt condition is not the only determinant of success. Some kernels are robust sanity checks across models and settings, while cipher and hash kernels expose semantic failures that feedback prompts do not repair reliably.

4.4 Task Difficulty and Case-Study Coverage

Table 4 groups failures by workload family. Statistical tests are the easiest group. Other block ciphers are the hardest. AES/components have high SL-policy pass but much lower functional pass, which is the clearest sign that security-shaped rewrites often damage exact cipher semantics. Hash kernels show the opposite caveat: high SL-policy pass can coexist with zero functional success on the harnessed hash task.

Table 4: Primary task-group aggregate across the eight stable model-condition rows. Columns are percentages. Syntax, SL-policy, and pre-HLS use the Rows denominator; Functional, Func+/SL-, and SL+/Func- use Func. n.

Task group	Rows	F.n	Syn.	Func.	SLP	Pre	F+/SL-	SL+/F-
AES/components	432	432	84.3%	34.0%	78.2%	28.2%	5.8%	50.0%
Other ciphers	96	96	63.5%	8.3%	17.7%	5.2%	3.1%	12.5%
Hash kernels	96	48	59.4%	0.0%	82.3%	22.9%	0.0%	97.9%
Statistical tests	192	192	90.1%	73.4%	62.5%	51.0%	22.4%	11.5%

Table 5: NTT/AutoNTT extension results. Each model-condition triplet is Syntax / SL-policy / Both over 36 rows (six NTT/AutoNTT tasks times six one-call settings); the Total row uses 144 rows per reasoning mode.

	Reasoning disabled			Reasoning enabled		
Model	Syn.	SL	Both	Syn.	SL	Both
Gemma4	0	16	0	7	29	5
DS-V4-Flash	0	25	0	1	35	1
DS-V4-Pro	3	26	3	3	31	3
Qwen3.5-122B-A10	0	24	0	1	29	1
Total	3	91	3	12	124	10

The task-level spread is large enough that a single aggregate would be misleading. `add_round_key` reaches 41/48 pre-HLS pass and `monobit` reaches 38/48, while `des` and `sha256_wip` reach 0/48. NTT/AutoNTT rows are kept separate: they add 36 source-level case-study rows per model condition, but they do not yet have the same project-local functional and HLS-wrapper coverage as the 17 primary tasks.

The NTT/AutoNTT extension is therefore read as a G1/G3 source-level integration stress test, not as an all-gate evaluation. Table 5 shows that across 288 extension rows, 215 pass the SL-policy checker, but only 15 pass syntax and only 13 pass both syntax and SL-policy. Reasoning-enabled runs improve this extension from 3 to 10 syntax-and-policy passes, but the absolute rate remains low because most generated candidates fail before functional or HLS validation can be meaningfully applied. This is a different failure mode from the 17 primary tasks: the security policy can be satisfied at the source-pattern level, but the model often fails to preserve the project-specific HLS/TAPA interface and dependency structure. We therefore include NTT/AutoNTT in the paper as extension evidence, while keeping it out of the primary pre-HLS/HLS denominators.

4.5 HLS, GateRepair, and PPA

For completed rows, HLS synthesis is usually not the bottleneck after pre-HLS pass: Gemma4 synthesizes 33/34 and 33/33 attempted candidates, DeepSeek Flash 27/28 and 33/34, DeepSeek Pro 23/23 and 42/44, and Qwen3.5-122B-A10 32/33 and 40/46. The harder downstream issue is PPA. We use the matched Gemma4 reasoning-disabled six-pilot reports only as exemplars, not as a full PPA distribution or model leaderboard. Table 6 keeps the small set of cases needed for the main claim: even when a candidate is executable,

Table 6: PPA examples normalized against original kernels.

Case	Example	Lat. ratio	LUT ratio	Main point
Easy rows	`add_round_key`, `monobit`	1.00×	1.00–1.25×	Near baseline
One-call S-box rewrite	`sub_bytes` Strategy/Semantic	215.68×	2.13–2.14×	Latency cost
GateRepair S-box rewrite	`sub_bytes` iter2	0.95×	53.58×	Area cost
Table AES rewrite	`aes_table` Semantic-feedback	178.16×	0.54×	Latency outlier

policy-compliant, and synthesizable, the resulting microarchitecture can move latency or area by orders of magnitude.

The takeaway is that easy rows stay near baseline, but table-heavy AES rows expose two PPA failure modes: single-shot security-guided rewrites can inflate latency by 178–216×, while GateRepair can restore latency by spending area, as in `sub_bytes` at 0.95× latency but 53.58× LUT. Low-latency rows that fail functional testing are not counted as PPA wins.

GateRepair is reported only as a diagnostic case study because it uses multiple calls and a changing feedback trajectory. It is not a seventh same-cost setting. On the six-pilot all-gate target, GateRepair reaches 3/6 for Gemma4 and 3/6 for Qwen3.5 off/on. This is a small improvement over Gemma4's best one-call result on the same six tasks (2/6), but it only matches the best one-call result for Qwen3.5. The successful repairs are the simpler `sub_bytes`, `add_round_key`, and `monobit` cases; `present`, `aes_table`, and `aes_tableless` remain hard. On the 17-primary SL-policy-only target, completed DeepSeek GateRepair runs reach 6/17, 8/17, and 7/17, which does not beat the corresponding best one-call SL-policy results. Thus the current GateRepair evidence is a negative finding: iterative feedback can repair simple cases, but it does not yet overcome complex cipher semantics or dominate the one-call settings.

5 Discussion

The preliminary evidence shows that security-constrained LLM-HLS evaluation is executable and measurable. The task packages, policy overlay, deterministic validators, prompt/feedback conditions, and HLS synthesis flow all operate at the primary workload scale. The same evidence also shows that the problem is not solved: final all-gate rates remain low, functional failure is usually the largest blocker, and policy-compliant rewrites can carry substantial PPA costs. The main lesson is methodological as much as quantitative: security-related LLM-HLS outputs need separated functional, source-level, synthesis, and PPA evidence rather than a single pass/fail label.

The SL-policy gate is a source-level validator, not a formal hardware security proof. It uses conservative source patterns and task policies to catch security-relevant flows that ordinary tests and synthesis do not reject, but it does not model all interprocedural, multi-hop, RTL, timing, or physical side-channel behavior. Future work should strengthen those validators and add larger security-labeled task sets; the evaluation structure here is intended to make those stronger checks comparable rather than replace them.

6 Conclusion

This paper evaluates current LLM capability on security-related HLS C/C++ generation and repair. The results show that ordinary HLS evaluation is necessary but incomplete for security-sensitive accelerators: syntax, functional tests, source-level policy checking, HLS synthesis, and PPA capture different evidence levels. A security-policy overlay and deterministic multi-gate validation make the direction measurable, and the preliminary Gemma4, DeepSeek, and Qwen3.5 results show both feasibility and substantial remaining failure modes. The NTT/AutoNTT extension shows the same G1/G3 difficulty outside the primary pool. GateRepair is useful diagnostically but currently repairs simple cases only, so stronger repair must preserve algorithmic semantics while removing flagged flows.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under Awards CNS-2348733, CNS-2505747, and CNS-2247892. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Stefan Abi-Karam and Cong Hao. 2025. HLS-Eval: A Benchmark and Framework for Evaluating LLMs on High-Level Synthesis Design Tasks. arXiv:2504.12268 doi:10.48550/arXiv.2504.12268

[2] Qirui Chen, Jingxian Shuai, Shuangwu Chen, Shenghao Ye, Zijian Wen, Xufei Su, Jie Jin, Jiangming Li, Jun Chen, Xiaobin Tan, and Jian Yang. 2026. HardSecBench: Benchmarking the Security Awareness of LLMs for Hardware Code Generation. arXiv:2601.13864 doi:10.48550/arXiv.2601.13864

[3] Fanghao Fan, Yingjie Xia, and Li Kuang. 2025. SecV: LLM-based Secure Verilog Generation with Clue-Guided Exploration on Hardware-CWE Knowledge Graph. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*. 8049–8057. doi:10.24963/ijcai.2025/895

[4] Ziteng Hu, Yingjie Xia, Xiyuan Chen, and Li Kuang. 2025. SecFSM: Knowledge Graph-Guided Verilog Code Generation for Secure Finite State Machines in Systems-on-Chip. arXiv:2508.12910 doi:10.48550/arXiv.2508.12910

[5] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. 2018. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 88. doi:10.1145/3240765.3243415

[6] M. Zafir Sadik Khan, Kimia Zamiri Azar, and Hadi Kamali. 2026. Bench4HLS: End-to-End Evaluation of LLMs in High-Level Synthesis Code Generation. arXiv:2601.19941 doi:10.48550/arXiv.2601.19941

[7] James Kuban and Tosiron Adegbija. 2023. Designing Constant-Timed Accelerators Using High-Level Synthesis: A Case Study of ECG Biometric Authentication. In *2023 International VLSI Symposium on Technology, Systems and Applications*. IEEE. doi:10.1109/VLSI-TSA/VLSI-DAT57221.2023.10134115

[8] Runkai Li, Jia Xiong, Xiuyuan He, Jieru Zhao, Qiang Xu, and Xi Wang. 2025. ChatHLS: Towards Systematic Design Automation and Optimization for High-Level Synthesis. arXiv:2507.00642 doi:10.48550/arXiv.2507.00642

[9] Runkai Li, Jia Xiong, and Xi Wang. 2025. iDSE: Navigating Design Space Exploration in High-Level Synthesis Using LLMs. arXiv:2505.22086 doi:10.48550/arXiv.2505.22086

[10] Nowfel Mashnoor, Mohammad Akyash, Hadi Kamali, and Kimia Zamiri Azar. 2025. TimelyHLS: LLM-Based Timing-Aware and Architecture-Specific FPGA HLS Optimization. arXiv:2507.17962 doi:10.48550/arXiv.2507.17962

[11] Ali Emre Oztas and Mahdi Jelodari. 2024. Agentic-HLS: An Agentic Reasoning Based High-Level Synthesis System Using Large Language Models. arXiv:2412.01604 doi:10.48550/arXiv.2412.01604

[12] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2019. TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2019), 798–808. doi:10.1109/TCAD.2018.2834421

[13] Shang Shi, Nitin Pundir, Hadi Mardani Kamali, Mark Tehranipoor, and Farimah Farahmandi. 2023. SecHLS: Enabling Security Awareness in High-Level Synthesis. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. 585–590. doi:10.1145/3566097.3567926

[14] Hanyu Wang, Xinrui Wu, Zijian Ding, Su Zheng, Chengyue Wang, Neha Prakriya, Tony Nowatzki, Yizhou Sun, and Jason Cong. 2025. LLM-DSE: Searching Accelerator Parameters with LLM Agents. arXiv:2505.12188 doi:10.48550/arXiv.2505.12188

[15] Zeng Wang, Minghao Shao, Weimin Fu, Prithwish Basu Roy, Xiaolong Guo, Ramesh Karri, Muhammad Shafique, Johann Knechtel, and Ozgur Sinanoglu. 2026. HarmChip: Evaluating Hardware Security Centric LLM Safety via Jailbreak Benchmarking. arXiv:2604.17093 doi:10.48550/arXiv.2604.17093

[16] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024. Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models. arXiv:2407.03889 doi:10.48550/arXiv.2407.03889